CS 3630 : Assignment 3: Monte Carlo Localization

Deadlines:

- **Check point (Parts 2 and 3):** Thurs, Feb 5rd 11:55pm via T-Square.
- **Final Assignment Due (Parts 4 and 5)**: Thurs, Feb 12th 11:55pm via T-Square.
- Assignment has to be done in groups of **2**.
- Mention your **group member names** in the submission.

It is absolutely not allowed to share your source code with anyone in the class as well as to use code from the Internet. If you have any questions, ask on Piazza or in class. **Do not give out answers on Piazza!**

Localization

As we know, robots don't quite behave as expected. This makes it particularly difficult to execute complicated tasks. We are going to use the concept of **localization**, specifically Monte Carlo Localization, to determine the most likely location of the robot through the use of sensors. For us, this means the use of the infrared sensor on the Fluke to determine the range of an object in front of the Scribbler.

Given a known map of the environment, the objective is to see if we can determine where the robot is within the map as it moves around. To do this, we are going to be using a **particle filter**. A particle filter maintains a collection of possible locations for the robot, known as particles. Each iteration it uses both the motion model of the robot as well as sensor readings to update the particles.

Read more about Monte Carlo Localization here: http://en.wikipedia.org/wiki/Monte_Carlo_localization

Provided Code

We have provided two folders of code, one in MATLAB and one in Python. The Python code is used to communicate with the Fluke+Scribbler, send commands, and read sensor values. It will automatically log all the data needed for the localization script. The MATLAB code reads in the recorded data and runs the localization algorithm.

You'll need to implement a couple functions in the MATLAB code during this assignment.

1 Download the RVC toolbox (Checkpoint)

Peter Corke's textbook that we are using has a companion set of code called the RVC toolbox. This is used as the basis of the matlab code and is required to run the code. Make sure you have it set on your MATLAB path after it's installed.

You can find it here:
http://petercorke.com/Robotics_Toolbox.html

2 Create an Environment (Checkpoint)

It should be a hallway-like environment with at least two 90 degree turns. We recommend having white walls, e.g. using white sheets of paper, to get maximal returns from the IR sensor.

Once you have measured your environment, replicate it in MATLAB using the **SquareMap** class. It takes a matrix with each row defining the top left corner x, y, width, and height of a single rectangle. You can see a simple example at the top of **script01_MotionModel.m** in the MATLAB folder.

Write-up:

Take a picture of your real-world environment as well as a screen shot of the simulated replication in MATLAB and include it in your write-up.

3 Create a Motion Plan (Checkpoint)

The python script **logData.py** has been provided to run the robot given a set of commands, and automatically log the motor values and the range sensor. You'll need to edit this script to queue a set of motion commands (see logData.py for an included example).

To improve the experience with interfacing with the robot, we've provided a python class called **Scribbler2.py** that expands upon the basic serial commands. This should simplify sending basic movement commands to a single function call **sendMotors**, which takes, assuming the fluke is facing forward, left motor values and right motor values between -200 and 200, and a duration to run those commands in seconds.

The sensors are polled every 0.1 seconds and log the motor values and the sensors at that time. To do this, we provided a new function **getObstacle**, explained below:

- **getObstacle**: Sends a command to pulse the IR emitter and read back the number of pulses received by the sensor. The maximum value is **6400** pulses, while the minimum is **0**. The range of an obstacle from the robot is inversely proportional to this number. A close object is one that reflects all of the pulses, while a far away object is one that does not return any pulses at all. **Note:** This varies depending the material and color of the object, darker objects will absorb most, if not all, of the IR beams. This is why we suggest you cover a portion of your walls in white paper to maximize reflectivity (especially if they are dark).

Verify:

Verify that the logged file is correct.

4 Run Particle Filter without Sensor Data (Final Submission)

The first step of Monte Carlo localization is to update the position of each particle using a motion model of the robot. We have provided a new class called **Differential.m**, which is a differential drive robot class that works with Corke's MATLAB toolbox. First however, you'll need to implement a portion of the code in Differential.

Implement the update function

When the particle filter updates the position of each particle, it uses the information from the log file, that is the left and right motor values we logged, and calculates the twist of the robot. Given this twist, it then computes a new pose of the robot and returns the twist.

Take the equations of motion and implement them in the code. You'll also need to use the r value you computed to convert the motor values into linear and angular velocities.

Run script01_MotionModel.m

Modify script01 to use your data log and simulated environment. After you've implemented **update()**, you should run the script. It will generate a bunch of particles and initialize them at the starting location of the robot. Each step, the simulation will update the location of the robot based upon the next line in your log file. It will then also update each particle using the vehicle's dynamics. This is the **prediction** step of the particle filter.

Write-up:

Include a screenshot of your robot moving through your world with the motion model only particle filter.

Question: As you run this code, you should see the particles expand and move away from the actual robot location over time. Why do you think they behave this way? Provide an intuitive explanation.

5 Run Particle Filter with Sensor Data (Final Submission)

The next steps has us weight the particles based upon the likelihood that they agree with the sensor readings. This is known as the **weighting step**. Take a look at the **weight()** function in the **RangeSensor** class.

Implement the weight function

For each particle, you'll want to define a non-zero weight for it depending upon it's difference from a predicted measurement. Essentially we want to find:

$$z_{diff} = z_{actual} - z_{prediction}$$

where we use a prediction function h to generate a measurement given the current particle's pose x

$$z_{prediction} = h(x)$$

Then, we want to define a weight based upon the **likelihood** that the particle agrees with the data. Assumming a Gaussian density, this is calculated by

$$weight = exp(-0.5 * z_{diff}^2 * inv(L))$$

Run script02_SensorModel.m

Finally, you have all the steps to run full Monte Carlo localization. Run script02 and see if your particles behave differently from before. If everything is working well, you should see them converge on the robot's location when the robot moves close enough to the wall.

Write-up:

Include a screenshot of your robot moving through your world with the full particle filter running.

Question: What differences do you see in the behavior of the particles with the sensors? Provide some possible meanings behind the shape of the particle cloud. Include screenshots to justify your arguments.

6 Submit

Turn in your MATLAB code as well as your writeup to T-Square. Only one submission needed per team, but please include all team member's names on the document.